Me-MPK: Accelerating Krylov Subspace Solvers via Memory-efficient Matrix-Power Kernel

Haozhong Qiu
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
qiuhaozhong16@nudt.edu.cn

Chuanfu Xu, Jianbin Fang, Shengguo Li
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
{xuchuanfu, j.fang, nudtlsg}@nudt.edu.cn

Liang Deng, Jian Zhang, Zhe Dai
China Aerodynamic Research and Development Center
Mianyang, China
dengliang11@nudt.edu.cn, zhangjian@cardc.cn
daizhe_cardc@163.com

Yue Ding, Yue Wang, Zhimeng Han
College of Computer Science and Technology
National University of Defense Technology
Changsha, China
{dingyue, wangyue, hanzhimeng24}@nudt.edu.cn

Yonggang Che, Jie Liu
College of Computer Science and Technology
National University of Defense Technology
Changsha, China
{ygche, liujie}@nudt.edu.cn

2025 62nd ACM/IEEE Design Automation Conference (DAC) | 979-8-3315-0304-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/DAC63849.2025.11133082

***Abstract*—This paper focuses on optimizing the Matrix-Power Kernel (MPK), which relies on a series of Sparse Matrix-Vector multiplications (SpMVs) using the same sparse matrix. MPK is a crucial component of Krylov subspace methods for solving large sparse linear systems in various fields, including circuit simulations. MPK offers a potential for matrix reuse in cache, which can accelerate memory-bound sparse solvers. Additionally, many sparse matrices encountered in applications are symmetric, allowing us to reduce the memory footprint for SpMVs by half. However, reusing the matrix introduces data dependencies between subsequent SpMVs, and symmetric SpMVs can result in data conflicts during shared-memory parallelization. Previous research has often focused on either matrix reuse or symmetry, failing to leverage both aspects effectively.**

This paper proposes a unified, memory-efficient approach called *Me-MPK* that takes advantage of both cache reuse and matrix symmetry for MPK on shared-memory multi-core systems. We first introduce a *unified dependency graph* for a sparse matrix, which represents all potential data dependencies and conflicts. Next, we perform architecture-aware recursive partitioning on this graph to create subgraphs and formulate a separating subgraph that decouples all dependencies and conflicts among the subgraphs. These independent subgraphs are then scheduled for parallel execution of SpMV or symmetric SpMV in a specified order to optimize cache reuse. We apply Me-MPK in two s-Step Krylov subspace solvers, and our evaluations show that Me-MPK significantly outperforms the current state-of-the-art solutions, delivering an average speedup of up to 2.00X and 1.86X on X86 and ARM CPUs, respectively. As a result, we achieve overall speedup in the sparse solvers of up to 1.65X and 1.58X.

***Index Terms*—SpMV, matrix power kernel, cache reuse, matrix symmetry, Krylov subspace solvers**

I. Introduction and Related Work

The Matrix-Power Kernel (MPK) involves a sequence of Sparse Matrix-Vector multiplications (SpMVs) to compute $\mathbf{Ax}$, $\mathbf{A^2x}$,..., $\mathbf{A^sx}$ for a matrix $\mathbf{A}$, a vector $\mathbf{x}$, and a constant s [1]–[3]. MPK is a crucial and time-consuming step in solving large-scale sparse linear systems, eigenvalue problems, and similar tasks [4]. Optimizing MPK is vital to enhance simulation efficiency across various fields, including circuit simulation and power grid analysis.

Given that SpMV is heavily constrained by memory, numerous efforts have been made to optimize memory efficiency for an individual SpMV invocation [5]–[8]. MPK offers a potential to exploit temporal locality for a sequence of SpMVs by reusing the matrix elements in cache, instead of repeatedly loading them from main memory. Meanwhile, the sparse matrices in many scientific applications are often symmetric [9]–[11]. Exploiting symmetric SpMV can significantly reduce memory usage by only storing either the upper or lower triangular part of the sparse matrix. Therefore, by taking advantage of both matrix reuse and symmetry, there is a significant opportunity to develop memory-efficient MPK, which can help mitigate the 'memory wall' problem that often challenges sparse solvers.

However, reusing the matrix in cache creates data dependencies between successive SpMV invocations. In the case of symmetric SpMV, there are data conflicts when multiple threads attempt to update the output vector simultaneously. These data dependencies and conflicts complicate the efficient and scalable implementation of multi-threaded processing on shared-memory multi-core systems.

Previous implementations of MPK have primarily focused on reusing matrices to minimize the frequency of distributed communications and synchronizations for communication-avoiding or s-Step algorithms [12]–[15]. The motivation behind exploiting cache reuse in MPK is largely rooted in classic blocking strategies used in stencil computations [16]–[19]. These methods typically involve identifying the neighboring blocks necessary for MPK concerning a specific matrix power s. This often leads to performance bottlenecks due to either

This work was funded by the National Natural Science Foundation of China under Grant No. 62272474. Co-first author: Chuanfu Xu. Corresponding authors: Chuanfu Xu, Jianbin Fang, Shengguo Li and Yonggang Che

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 04,2025 at 14:55:14 UTC from IEEE Xplore. Restrictions apply.

redundant computations or irregular matrix accesses. A recent level-based cache blocking approach for MPK [2], based on the Recursive Algebraic Coloring Engine (RACE) [10], organizes matrix rows into levels using a distance-k coloring. This method maintains dependencies for cache reuse but limits parallelism, allowing only one level to be executed in parallel at a time. Therefore, it relies on point-to-point synchronization and level coarsening strategies to improve scalability, but its performance still degrades severely for large power. Zhang *et al.* partition a sparse matrix and create a computation pipeline for fine-grained cache reuse by reusing a matrix row rather than a row block in MPK [1]. Although the approach is easy to implement, it requires graph coloring for parallelization, which can result in high synchronization overhead and poor spatial locality [20]. To summarize, all MPK implementations overlook the symmetry of the matrix. Thus, our work aims to leverage both cache reuse and matrix symmetry to further enhance MPK performance.

There has been some work in implementing symmetric SpMV, primarily focused on efficiently managing data conflicts at the shared-memory level. For instance, a reduction-based approach [21], [22] mitigates conflicts by using thread-local vectors to store temporary results, followed by a reduction loop to accumulate these results into the output vector. However, this method increases memory usage and traffic, which grows linearly with the number of threads, thereby limiting its overall scalability. Recently, researchers have introduced graph coloring algorithms to identify independent row sets from the graph representation of the symmetric matrix for parallel execution [9], [10]. While coloring-based approaches demonstrate impressive performance for low-bandwidth symmetric matrices, they struggle to scale effectively with large, irregular, high-bandwidth matrices. Furthermore, existing implementations of symmetric SpMV typically focus on a single invocation and do not leverage cache reuse for MPK. In summary, prior work has not fully exploited on the advantages of both matrix reuse and matrix symmetry; they tend to address only one aspect or the other.

This paper optimizes MPK by introducing a new, unified and memory-efficient approach called *Me-MPK*, which takes advantage of both cache reuse and matrix symmetry for MPK on multi-cores. We present the *unified dependency graph* for a sparse matrix, where each vertex represents a row of the matrix, and each edge denotes potential data dependencies or conflicts between rows. We then apply architecture-aware recursive partitioning to this graph to create subgraphs. In each recursion, we formulate a separate subgraph, known as the *separator*, to eliminate all dependencies and conflicts among the subgraphs. All subgraphs are then reordered to enhance data locality. Finally, we schedule these independent subgraphs in a specific order for parallel execution, ensuring that the same subgraph can be reused in the cache at least twice for subsequent calls to SpMV or symmetric SpMV.

We apply Me-MPK in two *s*-Step Krylov subspace methods, i.e., the Conjugate Gradient (CG) method and the Biconjugate Gradient Stabilized (BiCGStab). Our work was evaluated on a 28-core Intel Xeon X86 CPU and a 32-core Kunpeng 920 ARMv8 CPU [23] with 214 sparse matrices from the SuiteSparse Matrix Collection [24]. We compare Me-MPK to the MPKs based on SpMV and symmetric SpMV from the vendor-tuned Intel oneMKL library [25] and the RACE approach [2]. On X86, Me-MPK outperforms the MPKs based on the oneMKL SpMV and symmetric SpMV by an average speedup of 1.95X and 1.56X for *s*=4, 2.00X and 1.59X for *s*=8. Compared to the RACE-based MPK, Me-MPK yields an average speedup of 1.32X and 1.20X for *s*=4, 1.42X and 1.26X for *s*=8 on X86 and ARM. As a consequence, Me-MPK accelerates the sparse solvers by an average speedup of up to 1.59X and 1.51X for *s*=4, 1.65X and 1.58X for *s*=8 on X86 and ARM, respectively.

II. Background

This section first introduces the background of SpMV and symmetric SpMV. Then we introduce the Matrix-Power Kernel (MPK) and its application in *s*-Step Krylov subspace solvers.

A. SpMV and Symmetric SpMV

The SpMV operation involves multiplying a sparse matrix $\mathbf{A}$ by a dense input vector $\mathbf{x}$ to produce a dense output vector $\mathbf{b}$. Sparse matrices are typically stored in a compressed format to conserve space, which involves only retaining their nonzero elements and their corresponding locations. The top of Fig.1(b) illustrates the popular Compressed Sparse Row (CSR) format for a symmetric sparse matrix $\mathbf{A}$, as shown in Fig.1(a). The CSR format uses two arrays: one for storing the nonzero values and another for their corresponding column indices. Additionally, a row pointer array indicates the number of nonzero elements in each row. Algorithm 1 (lines 1-8) presents the pseudo-code for the serial SpMV in CSR.

The most common storage format for symmetric sparse matrices is the Sparse Symmetric Skyline (SSS) format [4], which is a variant of CSR with an additional array to store the matrix's main diagonal. The bottom of Fig.1(b) shows the SSS format storing the upper triangular part of the matrix $\mathbf{A}$. Algorithm 1 (lines 9-18) shows the pseudo-code of serial symmetric SpMV in SSS. Shared-memory parallelization of the outer loop (line 10) will introduce race conditions due to concurrently writing the output vector $\mathbf{b}$ ($b[col]$ in line 15 and $b[i]$ in line 17). Fig.1(d) illustrates this problem for the symmetric matrix $\mathbf{A}$ using two threads. In this example, thread 0 not only updates entries b_0 to b_3 but also affects b_4 to b_6, leading to write conflicts between the two threads.

B. MPK and Its Application in Krylov Subspace Solvers

MPK calculates a sequence of SpMVs to obtain $\mathbf{Ax}$, $\mathbf{A^2x}$,... $\mathbf{A^sx}$ or their linear combination such as $\mathbf{Ax} + \mathbf{A^2x}$, for the same matrix $\mathbf{A}$. Algorithm 1 (lines 19-22) shows the pseudo-code for a traditional MPK implementation, which performs s consecutive calls to the SpMV function. If the caches are too small to hold the entire matrix, it will need to be loaded s times from the main memory.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 04,2025 at 14:55:14 UTC from IEEE Xplore. Restrictions apply.

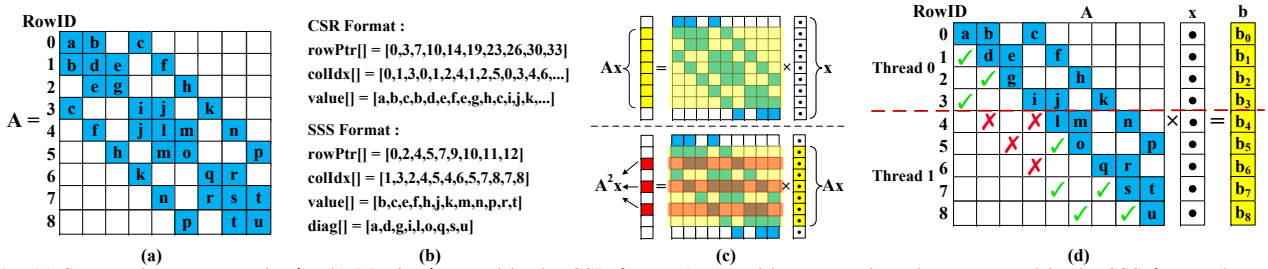


Fig. 1. (a) Symmetric sparse matrix $\mathbf{A}$; (b) Matrix $\mathbf{A}$ stored in the CSR format (top) and its upper triangular part stored in the SSS format (bottom); (c) Data reuse between $\mathbf{Ax}$ and $\mathbf{A^2x}$; (d) Data conflicts (red $\times$) between two threads using the upper triangular part of $\mathbf{A}$ in symmetric SpMV.

Algorithm 1: SpMV, symmetric SpMV and MPK

```
1  function SpMV(A, x, b)
2  for i ← 0 to N do
3      sum ← 0
4      for j ← A.rowPtr[i] to A.rowPtr[i + 1] do
5          sum = sum + A.value[j] · x[A.colIdx[j]]
6      end
7      b[i] = b[i] + sum
8  end
9  function SymmSpMV(A, x, b)
10 for i ← 0 to N do
11     sum ← A.diag[i] · x[i]
12     for j ← A.rowPtr[i] to A.rowPtr[i + 1] do
13         col = A.colIdx[j]
14         sum = sum + A.value[j] · x[col]
15         b[col] = b[col] + A.value[j] · x[i]
16     end
17     b[i] = b[i] + sum
18 end
19 function MPK(A, x, s)
20 for i ← 0 to s do
21     SpMV(A, x[:,i], x[:,i+1])
22 end
```

Algorithm 2: s-Step conjugate gradient method

```
1  Let A ∈ ℝ^{n×n} be a SPD matrix and x_0 be an initial guess.
2  Compute r_0 = b − Ax_0 and let index k = is (initially k = 0)
3  for i = 0, 1, ... until convergence do
4      Compute T = [r_k, Ar_k, ..., A^s r_k]        ▷ Matrix-power kernel
5      Let R_i = [r_k, Ar_k, ..., A^{s−1} r_k]      ▷ Extract R from T
6      Let P_i = [Ar_k, A^2 r_k, ..., A^{s−1} r_k]  ▷ Extract P from T
7      if i == 0 then                               ▷ Find directions p
8          Set P_i = R_i
9      else
10         Compute C_i = −Q_i^T P_{i−1}             ▷ Block dot-products
11         Solve W_{i−1} B_i = C_i                  ▷ Find scalars β
12         Compute P_i = R_i + P_i B_i
13     end
14     Compute W_i = Q_i^T P_i                      ▷ Block dot-products
15     Compute g_i = P_i^T r_i
16     Solve W_i a_i = g_i                          ▷ Find scalars α
17     Compute x_{k+s} = x_k + P_i a_i              ▷ Compute new x_k
18     Compute r_{k+s} = b − Ax_{k+s}               ▷ Recompute residual r_k
19     if ||r_{k+s}||_2 / ||r_0||_2 ≤ tol then stop end   ▷ Check convergence
20 end
```

MPK is a crucial component in many applications, such as Krylov subspace methods, multigrid methods, polynomial preconditioners, and eigenvalue problems. This work applies our MPK within two s-Step (or communication-avoiding) Krylov subspace methods: the Conjugate Gradient (CG) method for solving Symmetric Positive Definite (SPD) linear systems, and the Biconjugate Gradient Stabilized (BiCGStab) method for addressing non-SPD linear systems. Algorithm 2 provides the pseudo-code for the s-Step CG method [26][1]. These Krylov subspace solvers are widely used to solve large-scale sparse linear systems of transistor-level SPICE-like circuit simulations.

The s-step Krylov solvers generate s Krylov basis vectors at a time using MPK (line 4 in Algorithm 2), potentially reducing the communication cost of standard Krylov solvers by s. MPK also offers an opportunity to reuse the matrix elements in the cache. However, reusing cache data incurs dependencies between multiple SpMV invocations. Fig.1(c) illustrates this problem with $s = 2$. Suppose the submatrix (yellow rows) has been loaded into cache for $\mathbf{Ax}$, it is possible to reuse a subset (red rows) of this submatrix for $\mathbf{A^2x}$. However, this data reuse must fulfill the dependencies between successive SpMVs, i.e., the column indices of the subset must be in the range of the row indices of the original submatrix.

[1]For the pseudo-code of the s-Step BiCGStab method, please refer to [26]

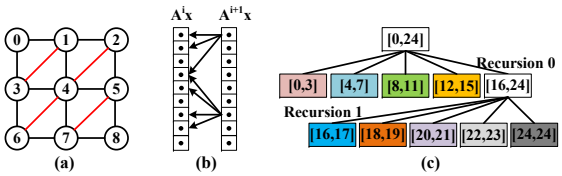


Fig. 2. (a) The unified dependency graph for the matrix in Fig.1(a); (b) Data dependencies between $\mathbf{A^ix}$ and $\mathbf{A^{i+1}x}$ for two vector elements; (c) The tree data structure corresponds to Fig.3(b). Numbers in the brackets indicate the starting and ending indices for matrix rows. [16,24] and [24,24] are the separators for the first and second recursions, respectively.

III. The Memory-efficient Matrix-Power Kernel

This section presents our unified approach to design memory-efficient MPK and its shared-memory parallelization.

A. Unified Dependency Graph

We propose *Unified Dependency Graph* (UDG), which is an undirected graph to represent all potential data dependencies and data conflicts in exploiting cache reuse and matrix symmetry for Me-MPK. Fig.2(a) shows the graph for the matrix $\mathbf{A}$ in Fig.1(a). Each vertex represents a matrix row. Each black edge indicates that a nonzero element of row i, which has a column index equal to j (or vice versa). These black edges correspond to all potential data dependencies among subsequent SpMV invocations. Fig.2(b) illustrates dependencies between two subsequent SpMVs for two vector elements: calculation of $\mathbf{A^{i+1}x}[0]$ and $\mathbf{A^{i+1}x}[6]$ depends on specific elements in $\mathbf{A^ix}$. When symmetric SpMV is employed in MPK, we introduce red edges in the graph, indicating that two rows have nonzero

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 04,2025 at 14:55:14 UTC from IEEE Xplore. Restrictions apply.

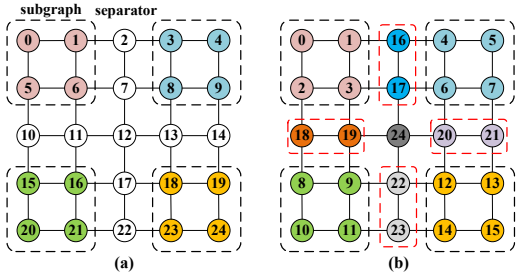


Fig. 3. (a) A graph partitioned into four subgraphs and one separator (without reordering); (b) After two recursive partitions (with reordering).

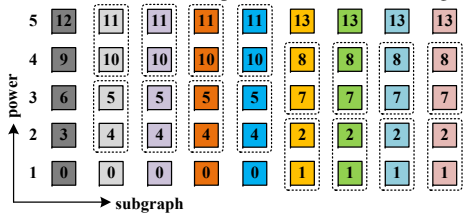


Fig. 4. Diagram illustrating the execution of Fig.3(b) in Me-MPK with the power *s=5*. Numbers in boxes indicate the execution order. Subgraphs with the same numbers are executed concurrently. Boxes in the same dotted rectangles indicate that subgraphs are reused in cache. For simplicity, this diagram assumes the sparse matrix is non-symmetric.

elements with the same column index. These black and red edges correspond to all potential data conflicts that may appear in a single invocation of shared-memory symmetric SpMV.

B. Architecture-aware Recursive Partitioning

For a given unified dependency graph of a sparse matrix, we propose an architecture-aware recursive approach to partition the graph. We start by partitioning the graph into *numPart* subgraphs. This can be implemented using existing graph partitioning tools like Metis [27]. For a CSR-based sparse matrix with 8-byte double values and 4-byte int indices, each subgraph contains no more than $partSize = \frac{cacheSize/8}{1.5*avg_nnz_per_row}$ rows, where $avg_nnz_per_row$ is the average number of nonzero elements per row, and *cacheSize* is the L2 cache size in bytes for the processor. This ensures that each subgraph can fit into the L2 cache. $numPart$ is set to $(\lfloor \frac{N/partSize}{numCore} \rfloor + 1) * numCore$, where N is the total number of matrix rows, allowing all subgraphs be evenly distributed among the *numCore* processor cores.

Then we formulate a special subgraph called *separator* by figuring out all vertices which have neighbors in other subgraphs. Next, we reorder vertices for each subgraph and the separator to ensure that each subgraph contains consecutive vertices, and the separator is indexed after the other subgraphs. This can improve the data locality when accessing the input/output vectors in SpMV. Fig.3 shows a graph after two recursive partitions (*numPart=4*). We use a tree pointer *treePtr* to save the result of recursive partitioning, used in the following parallel implementation. Fig.2(c) shows the tree data structure for Fig.3(b).

For invocations of standard SpMV in Me-MPK, we partition the separator one more time. For symmetric SpMV invocations, we recursively partition the separator multiple times. The optimal number of recursive partitions depends on the specific sparse symmetric matrices and the architectures used. This will be discussed further in the experiments.

Algorithm 3: Memory-efficient parallel MPK

```
1  function uSpMV(A, x, b, treePtr)
2  for i ← treePtr->st to treePtr->ed do
3      if A is symmetric then
4          line 11-17 in Algorithm 1
5      else
6          line 3-7 in Algorithm 1
7      end
8  end
9  function SymmSpMV_Recursive(A, x, b, treePtr)
10 if treePtr->separator == nullptr then
11     uSpMV(A, x, b, treePtr)
12     return
13 end
14 for i ← 0 to treePtr->numPatrs do in parallel
15     uSpMV(A, x, b, treePtr->son[i])
16 end
17 SymmSpMV_Recursive(A, x, b, treePtr->separator)
18 function puSpMV(A, x, b, treePtr)
19 if A is symmetric then
20     SymmSpMV_Recursive(A, x, b, treePtr)
21 else
22     for i ← treePtr->st to treePtr->ed do in parallel
23         line 3-7 in Algorithm 1
24     end
25 end
26 function Me_MPK(A, x, s, treeRoot)
27 if s==1 then
28     puSpMV(A, x[:,0], x[:,1], treeRoot)
29     return
30 end
31 i ← 0
32 separator ← treeRoot->separator
33 puSpMV(A, x[:,0], x[:,1], separator)
34 while i < s do
35     for j ← 0 to treeRoot->numParts do in parallel
36         uSpMV(A, x[:,i], x[:,i+1], treeRoot->son[j])
37         if i + 1 < s then
38             uSpMV(A, x[:,i+1], x[:,i+2], treeRoot->son[j])
39         else stop; end
40     end
41     i ← i+2
42     if i < s then
43         puSpMV(A, x[:,i-1], x[:,i], separator->separator)
44         for j ← 0 to separator->numParts do in parallel
45             uSpMV(A, x[:,i-1], x[:,i], separator->son[j])
46             uSpMV(A, x[:,i], x[:,i+1], separator->son[j])
47         end
48         puSpMV(A, x[:,i], x[:,i+1], separator->separator)
49     end
50     else
51         puSpMV(A, x[:,i-1], x[:,i], separator)
52     end
53 end
```

C. Parallel Implementation

After recursive partitioning of the graph for a sparse matrix, we obtain independent subgraphs (i.e., submatrices) at each recursion, because all edges (i.e., data dependencies and data conflicts) among subgraphs have been removed via separators. These submatrices, represented by the starting and ending row indices in the tree data structure, can be scheduled for parallel execution in a specified order to achieve cache reuse in MPK.

Algorithm 3 presents the pseudo-code of Me-MPK. To facilitate the implementation of Me-MPK for both symmetric and non-symmetric sparse matrices, the serial SpMV and symmetric SpMV for a submatrix (row index between `treePtr->st` and `treePtr->ed`) are encapsulated in a

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 04,2025 at 14:55:14 UTC from IEEE Xplore. Restrictions apply.

TABLE I
DETAILED CONFIGURATIONS OF MULTI-CORE CPUS.

	Xeon Gold 6258R	Kunpeng 920
Core frequency	2.4 GHz	2.6 GHz
Core number	28	32
L1 cache (I/D)	32/32 KiB	64/64 KiB
L2 cache (per core)	1024 KiB	512 KiB
L3 cache	39 MiB	32 MiB
Main memory	128 GiB	128 GiB

unified kernel `uSpMV` (lines 1-8). Accordingly, the parallel version is encapsulated in another unified kernel `puSpMV` (lines 18-25). Note that for standard SpMV, shared-memory parallelization can be directly implemented at the row loop (line 22); for symmetric SpMV, we implement a recursive kernel `SymmSpMV_Recursive` (lines 9-17) to parallelize it at the subgraph loop (line 14). This can avoid data conflicts for shared-memory parallelization of symmetric SpMV.

Me-MPK is implemented using two kernels: `uSpMV` and `puSpMV`. Fig.4 shows the execution order for different subgraphs shown in Fig.3(b). Since each subgraph or submatrix can fit into the L2 cache and all data dependencies or conflicts have been addressed, we can immediately schedule SpMVs or symmetric SpMVs for subsequent powers to use the entire submatrix that has been loaded into L2 from the previous power. For example, the subgraphs marked with an execution order of 1 and those marked with an execution order of 2 are reused in L2, as indicated by the dotted rectangles. Fig.4 shows that most subgraphs, except the separator at the second recursion, can be reused successfully.

IV. PERFORMANCE EVALUATION

This section begins by introducing our experimental setup, which includes the hardware, software, and tested matrices. We then present and analyze the performance results.

A. Experiment Setup

Hardware and software. We perform experiments on two multi-core CPUs: one Intel Xeon Gold 6258R X86 CPU@2.4GHz and one HUAWEI Kunpeng 920 ARMv8 CPU@2.6GHz [23]. Table I shows the detailed configurations for each CPU. All codes are compiled in double precision with '-O3' using Intel oneAPI 2022.1 for the X86 architecture and GCC 8.4.1 for the ARM architecture, respectively.

We set thread affinity for all tests, binding each software thread to an individual physical core (without hyperthreading). We run different implementations 100 times for each matrix and measure the average GFLOPS.

Sparse matrices. We select 214 large matrices from the SuiteSparse Matrix Collection [24] for our tests, where each matrix has more than 100,000 rows and more than 200,000 non-zero elements. Out of these, 162 sparse matrices are numerically symmetric. Additionally, we chose 12 circuit matrices (Table II) for a detailed comparison of our approach with existing methods. For non-symmetric matrices, we perform two rounds of partitioning. For symmetric matrices, we profile each matrix across two CPUs and conduct no more than four rounds of partitioning for most of them. The preprocessing cost associated with recursive partitioning for a sparse matrix is generally less than the time taken for one solver iteration.

TABLE II
THE TESTED CIRCUIT MATRICES. SPD INDICATES WHETHER A MATRIX IS SYMMETRIC POSITIVE DEFINITE. N IS THE ROW NUMBER. NNZ IS THE NON-ZERO ELEMENT NUMBER.

ID	Circuit matrix	N	NNZ	SPD
1	ASIC_320k	321821	1931828	no
2	ASIC_320ks	321671	1316085	no
3	ASIC_680ks	682712	1693767	no
4	Freescale1	3428755	17052626	no
5	FullChip	2987012	26621983	no
6	G2_circuit	150102	726674	yes
7	G3_circuit	1585478	7660826	yes
8	memchip	2707524	13343948	no
9	nxp1	414604	2656177	no
10	pre2	659033	5834044	no
11	rajat30	643994	6175244	no
12	transient	178866	961790	no

TABLE III
AVERAGE SPEEDUPS OF VARIOUS MPKS USING MKL ON X86 AND CSR ON ARM AS BASELINES.

	Intel 6258			Kunpeng 920		
	s=1	*s=4*	*s=8*	*s=1*	*s=4*	*s=8*
MKL-symm	1.27	1.23	1.23	-	-	-
RACE	1.10	1.43	1.35	1.07	1.52	1.47
Me-MPK(*reuse*)	1.17	1.53	1.59	1.21	1.51	1.54
Me-MPK(*reuse+symm*)	1.69	1.95	2.00	1.60	1.84	1.87

B. Performance Results

1) Overall Performance: On the X86 CPU, we evaluate the following MPK implementations: ❶ MKL. This version calls the Intel oneMKL SpMV in MPK, without exploiting cache reuse and matrix symmetry; ❷ MKL-symm. This version calls the Intel oneMKL symmetric SpMV to exploit matrix symmetry in MPK; ❸ RACE. This version is implemented by [2], which exploits cache reuse in MPK using the Recursive Algebraic Coloring Engine (RACE) approach [10]; ❹ Me-MPK(*reuse*). This version calls the standard CSR-based SpMV, but implements our strategy of cache reuse in MPK; ❺ Me-MPK(*reuse+symm*). This version implements our strategy of cache reuse, and it also calls our symmetric SpMV to exploit matrix symmetry in MPK. On the ARM CPU, we compare our work with MPK based on the standard SpMV using CSR format (referred to as CSR) and MPK based on RACE (referred to as RACE).

Fig.5 and Fig.6 show the overall performance for different MPK versions on the two CPU processors. Table III lists the average speedups of various MPKs using MKL on X86 and CSR on ARM as baselines. Compared to other MPKs, our two MPK versions, i.e., Me-MPK(*reuse*) and Me-MPK(*reuse+symm*), achieve higher speedups on both CPU platforms. Among all MPKs, Me-MPK(*reuse+symm*) performs best: On X86, it improves the performance by 1.95X and 2.00X, 1.56X and 1.60X, 1.32X and 1.42X, compared to MPKs based on oneMKL SpMV, oneMKL symmetric SpMV and RACE, for *s*=4 and *s*=8; On ARM, it improves the performance by 1.84X and 1.87X, 1.20X and 1.26X, compared to MPKs based on CSR and RACE, for *s*=4 and *s*=8. Compared to Me-MPK(*reuse*), Me-MPK(*reuse+symm*) enhances the performance by 1.32X and 1.21X, 1.14X and 1.19X on X86 and ARM for *s*=4 and *s*=8. These results

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 04,2025 at 14:55:14 UTC from IEEE Xplore. Restrictions apply.

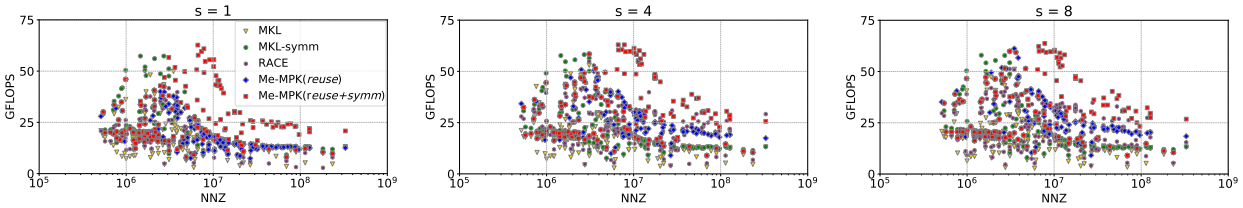


Fig. 5. Performance in GFLOPS for different MPK implementations using 214 sparse matrices on the Intel 6258R CPU. Note that MKL-symm and Me-MPK(*reuse+symm*) are evaluated using 162 symmetric matrices.

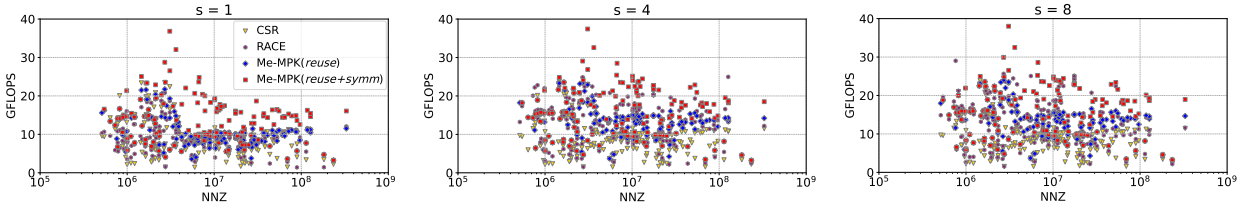


Fig. 6. Performance in GFLOPS for different MPK implementations using 214 sparse matrices on the Kunpeng 920 CPU. Note that Me-MPK(*reuse+symm*) is evaluated using 162 symmetric matrices.

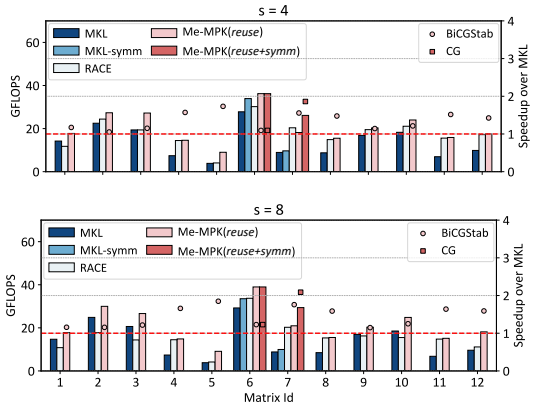


Fig. 7. Performance in GFLOPS for different MPK implementations using the 12 circuit matrices on the Intel 6258R CPU. The right Y-axis indicates the speedup for solvers compared to MKL-based baseline.

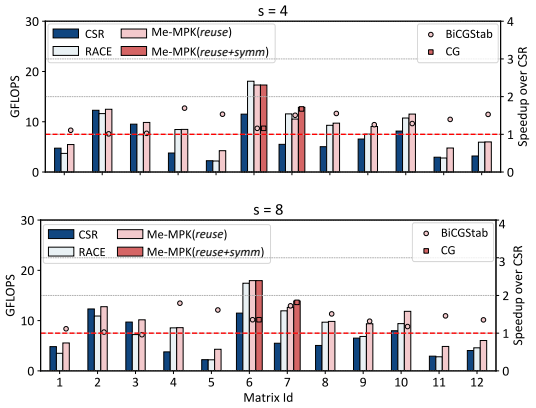


Fig. 8. Performance in GFLOPS for different MPK implementations using the 12 circuit matrices on the Kunpeng 920 CPU. The right Y-axis indicates the speedup for solvers compared to CSR-based baseline.

demonstrate the advantage of exploiting both matrix reuse and matrix symmetry in MPK. Note that the performance of RACE-based MPK drops for large power s. This is mainly because RACE reuses matrix elements in the L3 cache where s is the reuse distance, and increasing s will probably exceed the cache capacity, making it unable to reuse the data [2]. On the contrary, our approach always reuses data in the L2 cache for two successive SpMVs (i,e., $\mathbf{A^i x}$ and $\mathbf{A^{i+1}x}$), and the number of total cache reuse will increase with s because we reuse data (s-*1*) times for a given power s.

We also evaluate Me-MPK in two s-Step Krylov subspace solvers (i.e., CG and BiCGStab). Compared to MKL-based solvers on X86 and CSR-based solvers on ARM, Me-MPK accelerates the performance for a single solver iteration by an average speedup of 1.76X and 1.60X (s=4), 1.81X and 1.62X(s=8) for CG. The average speedup for BiCGStab is 1.34X and 1.33X (s=4), 1.40X and 1.38X (s=8) on X86 and ARM, respectively. Note that our approach fulfills all data dependencies and conflicts when exploiting matrix reuse and symmetry in MPK, therefore it has no negative effect on the solver convergence.

2) Performance on Circuit Matrices: Fig.7 and Fig.8 show the performance for the 12 circuit matrices. For 10 non-SPD matrices, Me-MPK(*reuse*) yields an average speedup of 1.57X and 1.61X, 1.18X and 1.38X, compared to MKL and RACE for s=4 and s=8 on X86; On ARM, the average speedup is 1.53X and 1.56X, 1.19X and 1.30X, compared to CSR and RACE for s=4 and s=8. For 2 SPD matrices, MPK(*reuse+symm*) delivers an average speedup of 1.95X and 2.11X, 1.70X and 1.85X, 1.24X and 1.30X, 1.17X and 1.18X, compared to MKL, MKL-symm, RACE and MPK(*reuse*)for s=4 and s=8 on X86; On ARM, the average speedup is 1.87X and 1.99X, 1.03X and 1.09X, 1.11X and 1.12X, compared to CSR, RACE and Me-MPK(*reuse*) for s=4 and s=8. Taking MKL-based solvers on X86 as baselines, we accelerate the s-Step CG and BiCGStab solvers by 1.63X and 1.40X (s=4), 1.65X and 1.41X (s=8) for circuit simulations. Taking CSR-based solvers on ARM as baselines, the average speedup is 1.49X and 1.38X, 1.59X and 1.40X for s=4 and s=8.

V. Conclusion

We have proposed Me-MPK, a memory-efficient Matrix-Power Kernel (MPK) designed for shared-memory multi-core systems. Me-MPK effectively leverages both cache reuse and matrix symmetry, significantly enhancing the performance of memory-bound sparse matrix solvers, such as the s-Step CG and BiCGStab methods that use MPK.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 04,2025 at 14:55:14 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Y. Zhang, S. Li, F. Yuan, D. Dong, X. Yang, T. Li, and Z. Wang, "Memory-aware optimization for sequences of sparse matrix-vector multiplications," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 379–389.

[2] C. Alappat, G. Hager, O. Schenk, and G. Wellein, "Level-based blocking for sparse matrices: Sparse matrix-power-vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 2, pp. 581–597, 2022.

[3] J. Gurhem, M. Vandromme, M. Tsuji, S. G. Petiton, and M. Sato, "Sequences of sparse matrix-vector multiplication on fugaku's a64fx processors," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 751–758.

[4] Y. Saad, *Numerical Methods for Large Eigenvalue Problems: Revised Edition*. SIAM, 2011.

[5] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on gpgpus," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 4, pp. 1–49, 2017.

[6] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994.

[7] R. W. Vuduc, *Automatic performance tuning of sparse matrix kernels*. University of California, Berkeley, 2003.

[8] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 2009, pp. 233–244.

[9] A. Elafrou, G. I. Goumas, and N. Koziris, "Conflict-free symmetric sparse matrix-vector multiplication on multicore architectures," *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[10] C. Alappat, A. Basermann, A. R. Bishop, H. Fehske, G. Hager, O. Schenk, J. Thies, and G. Wellein, "A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication," *ACM Trans. Parallel Comput.*, vol. 7, no. 3, jun 2020.

[11] H. Qiu, C. Xu, J. Fang, J. Zhang, L. Deng, Y. Ding, Q. Wang, S. Chen, Y. Che, and J. Liu, "A conflict-aware divide-and-conquer algorithm for symmetric sparse matrix-vector multiplication," in *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*. IEEE Computer Society, 2024, pp. 761–775.

[12] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in computing krylov subspaces," *EECS Dept., UC Berkeley, Tech. Rep. UCB/EECS-2007-123*, 2007.

[13] M. Hoemmen, *Communication-avoiding Krylov subspace methods*. University of California, Berkeley, 2010.

[14] E. C. Carson, *Communication-avoiding Krylov subspace methods in theory and practice*. University of California, Berkeley, 2015.

[15] J. Dongarra, S. Tomov, P. Luszczek, J. Kurzak, M. Gates, I. Yamazaki, H. Anzt, A. Haidar, and A. Abdelfattah, "With extreme computing, the rules have changed," *Computing in Science & Engineering*, vol. 19, no. 3, pp. 52–62, 2017.

[16] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–12.

[17] T. Muranushi and J. Makino, "Optimal temporal blocking for stencil computation," *Procedia Computer Science*, vol. 51, pp. 1303–1312, 2015.

[18] J. Morlan, S. Kamil, and A. Fox, "Auto-tuning the matrix powers kernel with sejits," in *High Performance Computing for Computational Science-VECPAR 2012: 10th International Conference, Kope, Japan, July 17-20, 2012, Revised Selected Papers 10*. Springer, 2013, pp. 391–403.

[19] D. Huber, M. Schreiber, and M. Schulz, "Graph-based multi-core higher-order time integration of linear autonomous partial differential equations," *Journal of Computational Science*, vol. 53, p. 101349, 2021.

[20] H. Qiu, C. Xu, J. Fang, L. Deng, J. Zhang, Q. Wang, Y. Ding, Z. Dai, Y. Che, S. Chen, and J. Liu, "Towards scalable unstructured mesh computations on shared memory many-cores," in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 109–119.

[21] V. H. F. Batista, G. O. Ainsworth, and F. L. B. Ribeiro, "Parallel structurally-symmetric sparse matrix-vector products on multi-core processors," *ArXiv*, vol. abs/1003.0952, 2010.

[22] T. Gkountouvas, V. P. Karakasis, K. Kourtis, G. I. Goumas, and N. Koziris, "Improving the performance of the symmetric sparse matrix-vector multiplication in multicore," *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 273–283, 2013.

[23] J. Xia, C. Cheng, X. Zhou, Y. Hu, and P. Chun, "Kunpeng 920: The first 7nm chiplet-based 64-core arm soc for cloud services," *IEEE Micro*, 2021.

[24] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1–25, 2011.

[25] Intel Corporation, *Intel oneAPI Math Kernel Library (oneMKL)*, Intel Corporation, 2024, accessed: March 25, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html

[26] M. Naumov, "S-step and communication-avoiding iterative methods," Technical Report NVR-2016-003, NVIDIA, Tech. Rep., 2016.

[27] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, 1998.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on November 04,2025 at 14:55:14 UTC from IEEE Xplore. Restrictions apply.